

1

INTRODUCTION

Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.

— MR. Cox **1893**, *Cinderella: Three hundred and forty-five variants* [?]]

TODO From the LIC

The first web browser, Nexus [?]], appeared in 1990. At that moment, web browsing was only about retrieving and showing small and static HTML web pages. In other words, users read the content of pages without interactions. The growing computing power of devices, the spread of the internet, and the need for more interaction and experiences for users encouraged the idea of executing code along with web pages. The Netscape browser made possible the execution of code on the client-side with the introduction of the JavaScript [?]] language in 1995. Remarkably, all browsers have supported JavaScript since Netscape. Nowadays, most web pages include not only HTML, but also include JavaScript code that is executed in client computers. During the past decades, web browsers have become JavaScript language virtual machines. They evolved to complex systems that can run full-fledged applications, like video and audio players, animation creators, and PDF document renderers such as the one showing this document.

JavaScript is currently the most used scripting language in all modern web browsers [?]]. However, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and recompiling the JavaScript code, which implies a significant overhead. Moreover, JavaScript faces security issues [?]]. For example, JavaScript lacks of memory isolation, making possible to extract pieces of information from others processes [?]]. Because of these problems, the Web Consortium (W3C) standardized in 2017 a bytecode for the web environment, the Wasm (Wasm) language.

Wasm is designed to be fast, portable, self-contained, and secure [?]]. All Wasm programs are compiled ahead-of-time from source languages such as

⁰Compilation probe time 2023/10/24 12:38:41

C/C++ and Rust. Wasm is created by third-party compilers that might include optimizations like in the case of LLVM. The Wasm language defines its Instruction Set Architecture [?] as an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, web browsers can use it to rapidly compile to the target architectures in a one-to-one translation process.

WebAssembly evolved outside web browsers. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [? ?]. Consequently, in 2019, the Bytecode Alliance [?] proposed WebAssembly System Interface (WASI) [?]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [?], making it suitable for edge-cloud computing platforms [? ?].

1.1 Software Monoculture

Web browsers and JavaScript have nearly three decades of development. Since then, web browsers have grown, reaching several implementations [? ?]. Nevertheless, only Firefox, Chrome, Safari, and Edge dominate on user computers. This means that, for 5 arbitrary devices (computers, tablets, smartphones) in a world of millions, at least two of them use the same web browser. This highlights a software monoculture problem [?], as an ecosystem of machines running the exact same software. The monoculture concept is an analogy from biology [?]. It describes an ecosystem that faces extinction due the lack of diversity as all individuals share the exact same vulnerabilities. In other words, many applications can crash due to a single shared vulnerability.

Nowadays, the serving of web pages, including WebAssembly code, is centralized and provided through main servers [?]. Thus, a similar argument for software monoculture can be used for the Wasm code that is served to web browsers. Despite being designed for sandboxing and secure execution, Wasm is not exempt from vulnerabilities [?]. For example, Wasm engines are vulnerable to speculative execution [?], and C/C++ source code vulnerabilities might be ported to Wasm binaries [?]. Therefore, the sharing of the Wasm code through web browsers, also includes Wasm vulnerabilities.

The software monoculture problem escalates if we consider the edge-cloud computing platforms and how they are adopting Wasm to provide services, as we previously mentioned. Concretely, along with browser clients, thousands of edge devices running Wasm as backend services might be affected due to vulnerabilities sharing. This means that if one node in an edge network is vulnerable, all the others are vulnerable in the exact same manner as the same binary is replicated on each node. In other words, the same attacker payload would break all edge nodes at once. This illustrates how Wasm execution is fragile with respect to systemic vulnerabilities for the whole internet. Let us take the example of what

happened on June 8, 2021, with Fastly [?]. That day, the whole internet suffered a 45 minutes disruption because of a failure when one Wasm binary was deployed at Fastly. The complete Fastly platform crashed. The bug, combined with most web pages being CDN-dependant, created a catastrophe. Therefore, a single distributed Wasm binary could unleash the same incident [?].

One might think that the solution is to adopt more web browser and interpreters implementations. However, this is virtually impossible as 4 web browsers dominate the market and edge-cloud computing platforms are transparently executed in the backend. Thus, a solution in this direction is doomed to fail. Another solution is to provide different WebAssembly codes. For example, a different source code, yet equivalent, can be provided when a web page requests it [?]. Consequently, millions of computers would execute different codes even though they use the same web browser. This strategy is called Software Diversification.

1.2 Software Diversification

Software Diversification is the process of finding, creating, and deploying program variants of a given original program [?] for the sake of security. Cohen et al. [?] and Forrest et al. [?] pioneered this field by proposing software diversification through code transformations. They proposed to produce variants of programs while preserving their functionalities, aiming to mitigate vulnerabilities. Since then, transformations aiming at reducing the predictability of observable behavior of programs have been proposed. For example, works on this direction proposed to diversify programs control flow [?], instruction set [?], or the system calls they use [?]. Several of these transformations can be combined to produce less predictable variants.

While previous works on software diversification demonstrated the removal of vulnerabilities, in all cases, it can be used as a preemptive solution. For example, if a vulnerability is present in one program variant, discovering and disseminating it will not affect other variants. Software diversification has been widely researched, yet, the field does not study its application to Wasm. Only Romano et al. [?] proposed the intermixing JavaScript and Wasm function calls to provide obfuscation against code analysis. For Wasm, no software diversification solution has been proposed, primarily due to its novelty.

TODO Recent papers first. Mention Workshops instead in conference. "Proceedings of XXXX". Add the pages in the papers list.

1.3 Background

TODO Motivate with the open challenges.

Contribution	Research papers			
	P1	P2	P3	P4
C1 Experimental contribution	✓	✓	✓	✓
C2 Theoretical contribution	✓		✓	
C3 Diversity generation	✓	✓	✓	✓
C4 Defensive diversification	✓	✓	✓	
C5 Offensive diversification				✓

Table 1.1

1.4 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.5 Automatic Software diversification requirements

1. 1: **TODO** Requirement 1

1.6 List of contributions

TODO Describe each one

C1 Experimental contribution: We propose reproducible methodology for generating software diversification for WebAssembly, the assessment of the generated diversity and the exploitation of the generated variants.

C2 Theoretical contribution: We propose a theoretical foundation in order to improve Software Diversification for WebAssembly.

C3 Diversity generation: We generate WebAssembly program variants.

C4 Defensive Diversification: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5 Offensive Diversification: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

1.7 Summary of research papers

This compilation thesis comprises the following research papers.

P1: CROW: Code randomization for WebAssembly bytecode.

Summary: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut

purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

P2: Multivariant execution at the Edge.

Summary: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut

purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

P3: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

Summary: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut

purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel

leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

P4: WebAssembly Diversification for Malware evasion.

Summary: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut

purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

■ Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author’s licentiate thesis [?]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.